Velisarios

Rahli, Vincent; Vukotic, Ivana; Völp, Marcus; Veríssimo, Paulo Jorge Esteves

DOI:
10.1007/978-3-319-89884-1_22

Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq

Vincent Rahli(✉), Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo

SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
{vincent.rahli,ivana.vukotic,marcus.voelp,paulo.verissimo}@uni.lu

Abstract. Our increasing dependence on complex and critical information infrastructures and the emerging threat of sophisticated attacks, ask for extended efforts to ensure the correctness and security of these systems. Byzantine fault-tolerant state-machine replication (BFT-SMR) provides a way to harden such systems. It ensures that they maintain correctness and availability in an application-agnostic way, provided that the replication protocol is correct and at least $n - f$ out of n replicas survive arbitrary faults. This paper presents Velisarios, a logic-of-events based framework implemented in Coq, which we developed to implement and reason about BFT-SMR protocols. As a case study, we present the first machine-checked proof of a crucial safety property of an implementation of the area's reference protocol: PBFT.

Keywords: Byzantine faults · State machine replication
Formal verification · Coq

1 Introduction

Critical information infrastructures such as the power grid or water supply systems assume an unprecedented role in our society. On one hand, our lives depend on the correctness of these systems. On the other hand, their complexity has grown beyond manageability. One state of the art technique to harden such critical systems is Byzantine fault-tolerant state-machine replication (BFT-SMR). It is a generic technique that is used to turn any service into one that can tolerate *arbitrary* faults, by extensively replicating the service to mask the behavior of a minority of possibly faulty replicas behind a majority of healthy replicas, operating in consensus.[1] The total number of replicas n is a parameter over the maximum number of faulty replicas f, which the system is configured to tolerate

This work is partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

[1] For such techniques to be useful and in order to avoid persistent and shared vulnerabilities, replicas need to be rejuvenated periodically [17,76], they need to be diverse enough [43], and ideally they need to be physically far apart. Diversity and rejuvenation are not covered here.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 619–650, 2018.
https://doi.org/10.1007/978-3-319-89884-1_22

at any point in time. Typically, $n = 3f + 1$ for classical protocols such as in [16], and $n = 2f + 1$ for protocols that rely on tamper-proof components such as in [82]. Because such protocols tolerate arbitrary faults, a faulty replica is one that does not behave according to its specification. For example it can be one that is controlled by an attacker, or simply one that contains a bug.

Ideally, we should guarantee the correctness and security of such replicated and distributed, hardened systems to the highest standards known to mankind today. That is, the proof of their correctness should be checked by a machine and their model refined down to machine code. Unfortunately, as pointed out in [29], most distributed algorithms, including BFT protocols, are published in pseudo-code or, in the best case, a formal but not executable specification, leaving their safety and liveness questionable. Moreover, Lamport, Shostak, and Pease wrote about such programs: "We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm." [54]. Therefore, we focus here on developing a generic and extensible formal verification framework for systematically supporting the mechanical verification of BFT protocols and their implementations.[2]

Our framework provides, among other things, a model that captures the idea of arbitrary/Byzantine faults; a collection of standard assumptions to reason about systems with faulty components; proof tactics that capture common reasoning patterns; as well as a general library of distributed knowledge. All these parts can be reused to reason about any BFT protocol. For example, most BFT protocols share the same high-level structure (they essentially disseminate knowledge and vote on the knowledge they gathered), which we capture in our knowledge theory. We have successfully used this framework to prove a crucial safety property of an implementation of a complex BFT-SMR protocol called PBFT [14–16]. We handle all the functionalities of the base protocol, including garbage collection and view change, which are essential in practical protocols. Garbage collection is used to bound message logs and buffers. The view change procedure enables BFT protocols to make progress in case the *primary*—a distinguished replica used in some fault-tolerant protocols to coordinate votes—becomes faulty.

Contributions. Our contributions are as follows: (1) Section 3 presents Velisarios, our continuing effort towards a generic and extensible logic-of-events based framework for verifying implementations of BFT-SMR protocols using Coq [25]. (2) As discussed in Sect. 4, our framework relies on a library to reason about *distributed epistemic knowledge*. (3) We implemented Castro's landmark PBFT protocol, and proved its agreement safety property (see Sect. 5). (4) We implemented a runtime environment to run the OCaml code we extract from Coq (see Sect. 6). (5) We released Velisarios and our PBFT safety proof under an open source licence.[3]

[2] Ideally, both (1) the replication mechanism and (2) the instances of the replicated service should be verified. However, we focus here on (1), which has to be done only once, while (2) needs to be done for every service and for every replica instance.

[3] Available at: https://github.com/vrahli/Velisarios.

Why PBFT? We have chosen PBFT because several BFT-SMR protocols designed since then either use (part of) PBFT as one of their main building blocks, or are inspired by it, such as [6,8,26,45,46,82], to cite only a few. Therefore, a bug in PBFT could imply bugs in those protocols too. Castro provided a thorough study of PBFT: he described the protocol in [16], studied how to proactively rejuvenate replicas in [14], and provided a pen-and-paper proof of PBFT's safety in [15,17]. Even though we use a different model—Castro used I/O automata (see Sect. 7.1), while we use a logic-of-events model (see Sect. 3)—our mechanical proof builts on top of his pen-and-paper proof. One major difference is that here we verify actual running code, which we obtain thanks to Coq's extraction mechanism.

2 PBFT Recap

This section provides a rundown of PBFT [14–16], which we use as running example to illustrate our model of BFT-SMR protocols presented in Sect. 3.

2.1 Overview of the Protocol

We describe here the public-key based version of PBFT, for which Castro provides a formal pen-and-paper proof of its safety. PBFT is considered the first practical BFT-SMR protocol. Compared to its predecessors, it is more efficient and it does not rely on unrealistic assumptions. It works with asynchronous, unreliable networks (i.e., messages can be dropped, altered, delayed, duplicated, or delivered out of order), and it tolerates independent network failures. To achieve this, PBFT assumes strong cryptography in the form of collision-resistant digests, and an existentially unforgeable signature scheme. It supports any deterministic state machine. Each state machine replica maintains the service state and implements the service operations. Clients send requests to all replicas and await $f + 1$ matching replies from different replicas. PBFT ensures that healthy replicas execute the same operations in the same order.

To tolerate up to f faults, PBFT requires $|R| = 3f+1$ replicas. Replicas move trough a succession of configurations called *views*. In each view v, one replica ($p = v \ mod \ |R|$) assumes the role of *primary* and the others become *backups*. The primary coordinates the votes, i.e., it picks the order in which client requests are executed. When a backup suspects the primary to be faulty, it requests a view-change to select another replica as new primary.

Normal-Case. During normal-case operation, i.e., when the primary is not suspected to be faulty by a majority of replicas, clients send requests to be executed, which trigger agreement among the replicas. Various kinds of messages have to be sent among clients and replicas before a client knows its request has been executed. Figure 1 shows the resulting message patterns for PBFT's normal-case operation and view-change protocol. Let us discuss here normal-case operation:

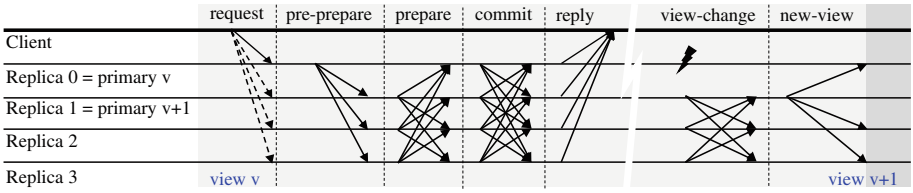

Fig. 1. PBFT normal-case (left) and view-change (right) operations

1. *Request:* To initiate agreement, a client c sends a request of the form $\langle \texttt{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the primary, but is also prepared to broadcast it to all replicas if replies are late or primaries change. $\langle \texttt{REQUEST}, o, t, c \rangle_{\sigma_c}$ specifies the operation to execute o and a timestamp t that orders requests of the same client. Replicas will not re-execute requests with a lower timestamp than the last one processed for this client, but are prepared to resend recent replies.
2. *Pre-prepare:* The primary of view v puts the pending requests in a total order and initiates agreement by sending $\langle \texttt{PRE-PREPARE}, v, n, m \rangle_{\sigma_p}$ to all the backups, where m should be the n^{th} executed request. The strictly monotonically increasing and contiguous sequence number n ensures preservation of this order despite message reordering.
3. *Prepare:* Backup i acknowledges the receipt of a pre-prepare message by sending the digest d of the client's request in $\langle \texttt{PREPARE}, v, n, d, i \rangle_{\sigma_i}$ to all replicas.
4. *Commit:* Replica i acknowledges the reception of $2f$ prepares matching a valid pre-prepare by broadcasting $\langle \texttt{COMMIT}, v, n, d, i \rangle_{\sigma_i}$. In this case, we say that the message is *prepared* at i.
5. *Execution & Reply:* Replicas execute client operations after receiving $2f + 1$ matching commits, and follow the order of sequence numbers for this execution. Once replica i has executed the operation o requested by client c, it sends $\langle \texttt{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$ to c, where r is the result of applying o to the service state. Client c accepts r if it receives $f+1$ matching replies from different replicas.

Client and replica authenticity, and message integrity are ensured through signatures of the form $\langle m \rangle_{\sigma_i}$. A replica accepts a message m only if: (1) m's signature is correct, (2) m's view number matches the current view, and (3) the sequence number of m is in the water mark interval (see below).

PBFT buffers pending client requests, processing them later in batches. Moreover, it makes use of checkpoints and water marks (which delimit sequence number intervals) to limit the size of all message logs and to prevent replicas from exhausting the sequence number space.

Garbage Collection. Replicas store all correct messages that were created or received in a log. Checkpoints are used to limit the number of logged messages by removing the ones that the protocol no longer needs. A replica starts checkpointing after executing a request with a sequence number divisible by some predefined constant, by multicasting the message $\langle \texttt{CHECKPOINT}, v, n, d, i \rangle_{\sigma_i}$ to all

other replicas. Here n is the sequence number of the last executed request and d is the digest of the state. Once a replica received $f + 1$ different checkpoint messages[4] (possibly including its own) for the same n and d, it holds a proof of correctness of the log corresponding to d, which includes messages up to sequence number n. The checkpoint is then called *stable* and all messages lower than n (except view-change messages) are pruned from the log.

View Change. The view change procedure ensures progress by allowing replicas to change the leader so as to not wait indefinitely for a faulty primary. Each backup starts a timer when it receives a request and stops it after the request has been executed. Expired timers cause the backup to suspect the leader and request a view change. It then stops receiving normal-case messages, and multicasts $\langle \texttt{VIEW-CHANGE}, v+1, n, s, C, P, i \rangle_{\sigma_i}$, reporting the sequence number n of the last stable checkpoint s, its proof of correctness C, and the set of messages P with sequence numbers greater than n that backup i prepared since then. When the new primary p receives $2f+1$ view-change messages, it multicasts $\langle \texttt{NEW-VIEW}, v+1, V, O, N \rangle_{\sigma_p}$, where V is the set of $2f + 1$ valid view-change messages that p received; O is the set of messages prepared since the latest checkpoint reported in V; and N contains only the special *null* request for which the execution is a no-op. N is added to the O set to ensure that there are no gaps between the sequence numbers of prepared messages sent by the new primary. Upon receiving this new-view message, replicas enter view $v + 1$ and re-execute the normal-case protocol for all messages in $O \cup N$.

We have proved a critical safety property of PBFT, including its garbage collection and view change procedures, which are essential in practical protocols. However, we have not yet developed generic abstractions to specifically reason about garbage collection and view changes, that can be reused in other protocols, which we leave as future work.

2.2 Properties

PBFT with $|R| = 3f+1$ replicas is safe and live. Its safety boils down to linearizability [42], i.e., the replicated service behaves like a centralized implementation that executes operations atomically one at a time. Castro used a modified version of linearizability in [14] to deal with faulty clients. As presented in Sect. 5, we proved the crux of this property, namely the agreement property (we leave linearizability for future work).

As informally explained by Castro [14], assuming weak synchrony (which constrains message transmission delays), PBFT is live, i.e., clients will eventually receive replies to their requests. In the future, we plan to extend Velisarios to support liveness and mechanize PBFT's liveness proof.

[4] Castro first required $2f + 1$ checkpoint messages [16] but relaxed this requirement in [14].

2.3 Differences with Castro's Implementation

As mentioned above, besides the normal-case operation, our Coq implementation of PBFT handles garbage collection, view changes and request batching. However, we slightly deviated from Castro's implementation [14], primarily in the way checkpoints are handled: we always work around sending messages that are not between the water marks, and a replica always requires its own checkpoint before clearing its log. Assuming the reader is familiar with PBFT, we now detail these deviations and refer the reader to [14] for comparison.

(1) To the best of our knowledge, to ensure liveness, Castro's implementation requires replicas to resend prepare messages below the low water mark when adopting a new-view message and processing the pre-prepares in $O \cup N$. In contrast, our implementation never sends messages with sequence numbers lower than the low water mark. This liveness issue can be resolved by bringing late replicas up to date through a state transfer.
(2) We require a new leader to send its own view-change message updated with its latest checkpoint as part of its new-view message. If not, it may happen that a checkpoint stabilizes after the view-change message is sent and before the new-view message is prepared. This might result in a new leader sending messages in $O \cup N$ with a sequence number below its low water mark, which it avoids by updating its own view-change message to contain its latest checkpoint.
(3) We require replicas to wait for their own checkpoint message before stabilizing a checkpoint and garbage collecting logs. This avoids stabilizing a checkpoint that has not been computed locally. Otherwise, a replica could lose track of the last executed request if its sequence number is superseded by the one in the checkpoint. Once proven, a state transfer of the latest checkpoint state and an update of the last executed request would also resolve this point.

We slightly deviated from Castro's protocol to make our proofs go through. We leave it for future work to formally study whether we could do without these changes, or whether they are due to shortcomings of the original specification.

3 Velisarios Model

Using PBFT as a running example, we now present our Coq model for Byzantine fault-tolerant distributed systems, which relies on a logic of events—Fig. 2 outlines our formalization.

3.1 The Logic of Events

We adapt the Logic of Events (LoE) we used in EventML [9,11,71] to not only deal with crash faults, but arbitrary faults in general (including malicious

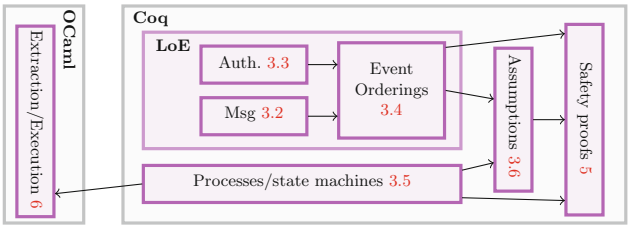

Fig. 2. Outline of formalization

faults). LoE, related to Lamport's notion of causal order [53] and to event structures [60,65], was developed to reason about events occurring in the execution of a distributed system. LoE has recently been used to verify consensus protocols [71,73] and cyber-physical systems [3]. Another standard model of distributed computing is Chandy and Lamport's *global state semantics* [19], where a distributed system is modeled as a single state machine: a state is the collection of all processes at a given time, and a transition takes a message in flight and delivers it to its recipient (a process in the collection). Each of these two models has advantages and disadvantages over the other. We chose LoE because in our experience it corresponds more closely to the way distributed system researchers and developers reason about protocols. As such, it provides a convenient communication medium between distributed systems and verification experts.

In LoE, an event is an abstract entity that corresponds either (1) to the handling of a received message, or (2) to some arbitrary activity about which no information is provided (see the discussion about trigger in Sect. 3.4). We use those arbitrary events to model arbitrary/Byzantine faults. An event happens at a specific point in space/time: the space coordinate of an event is called its location, and the time coordinate is given by a well-founded ordering on events that totally orders all events at the same location. Processes react to the messages that triggered the events happening at their locations one at a time, by transitioning through their states and creating messages to send out, which in turn might trigger other events. In order to reason about distributed systems, we use the notion of *event orderings* (see Sect. 3.4), which essentially are collections of ordered events and represent runs of a system. They are abstract entities that are never instantiated. Rather, when proving a property about a distributed system, one has to prove that the property holds for all event orderings corresponding to all possible runs of the system (see Sects. 3.5 and 5 for examples). Some runs/event orderings are not possible and therefore excluded through assumptions, such as the ones described in Sect. 3.6. For example, exists_at_most_f_faulty excludes event orderings where more than f out of n nodes could be faulty.

In the next few sections, we explain the different components (messages, authentication, event orderings, state machines, and correct traces) of Velisarios, and their use in our PBFT case study. Those components are parameterized by abstract types (parameters include the type of messages and the kind of authentication schemes), which we later have to instantiate in order to reason

about a given protocol, e.g. PBFT, and to obtain running code. The choices we made when designing Velisarios were driven by our goal to generate running code. For example, we model cryptographic primitives to reason about authentication.

3.2 Messages

Model. Some events are caused by messages of type msg, which is a parameter of our model. Processes react to messages to produce message/destinations pairs (of type DirectedMsg), called *directed messages*. A directed message is typically handled by a message outbox, which sends the message to the listed destinations.[5] A destination is the name (of type name, which is a parameter of our model) of a node participating in the protocol.

PBFT. In our PBFT implementation, we instantiate the msg type using the following datatype (we only show some of the normal-case operation messages, leaving out for example the more involved pre-prepare messages—see Sect. 2.1):

```
Inductive PBFTmsg :=                Inductive Bare_Prepare :=
| REQUEST (r : Request)             | bare_prepare (v : View) (n : SeqNum) (d : digest) (i : Rep).
| PREPARE (p : Prepare)             Inductive Prepare :=
| REPLY (r : Reply) ...             | prepare (b : Bare_Prepare) (a : list Token).
```

As for prepares, all messages are defined as follows: we first define bare messages that do not contain authentication tokens (see Sect. 3.3), and then authenticated messages as pairs of a bare message and an authentication token. Views and sequence numbers are nats, while digests are parameters of the specification. PBFT involves two types of nodes: replicas of the form PBFTreplica(r), where r is of type Rep; and clients of the form PBFTclient(c), where c is of type Client. Both Rep and Client are parameters of our formalization, such that Rep is of arity 3f+1, where f is a parameter that stands for the number of tolerated faults.

3.3 Authentication

Model. Our model relies on an abstract concept of keys, which we use to implement and reason about authenticated communication. Capturing authenticity at the level of keys allows us to talk about impersonation through key leakage. Keys are divided into *sending keys* (of type sending_key) to authenticate a message for a target node, and *receiving keys* (of type receiving_key) to check the validity of a received message. Both sending_key and receiving_key are parameters of our model.[6] Each node maintains *local keys* (of type local_keys), which consists of two lists of *directed keys*: one for sending keys and one for receiving keys. Directed keys are pairs of a key and a list of node names identifying the processes that the holder of the key can communicate with.

[5] Message inboxes/outboxes are part of the runtime environment but not part of the model.

[6] Sending and receiving keys must be different when using asymmetric cryptography, and can be the same when using symmetric cryptography.

Sending keys are used to create *authentication tokens* of type Token, which we use to authenticate messages. Tokens are parameters of our model and abstract away from concrete concepts such as digital signatures or MACs. Typically, a message consists of some data plus some tokens that authenticates the data. Therefore, we introduce the following parameters: (1) the type data, for the kind of data that can be authenticated; (2) a create function to authenticate some data by generating authentication tokens using the sending keys; and (3) a verify function to verify the authenticity of some data by checking that it corresponds to some token using the receiving keys.

Once some data has been authenticated, it is typically sent over the network to other nodes, which in turn need to check the authenticity of the data. Typically, when a process sends an authenticated message to another process it includes its identity somewhere in the message. This identity is used to select the corresponding receiving key to check the authenticity of the data using verify. To extract this claimed identity we require users to provide a data_sender function.

It often happens in practice that a message contains more than one piece of authenticated data (e.g., in PBFT, pre-prepare messages contain authenticated client requests). Therefore, we require users to provide a get_contained_auth_data function that extracts all authenticated pieces of data contained in a message. Because we sometimes want to use different tokens to authenticate some data (e.g., when using MACs), an authenticated piece of data of type auth_data is defined as a pair of: (1) a piece of data, and (2) a list of tokens.

PBFT. Our PBFT implementation leaves keys and authentication tokens abstract because our safety proof is agnostic to the kinds of these elements. However, we turn them into actual asymmetric keys when extracting OCaml code (see Sect. 6 for more details). The create and verify functions are also left abstract until we extract the code to OCaml. Finally, we instantiate the data (the objects that can be authenticated, i.e., bare messages here), data_sender, and get_contained_auth_data parameters using:

```
Inductive PBFTdata := | PBFTdata_request (r : Bare_Request)
  | PBFTdata_prepare (p : Bare_Prepare) | PBFTdata_reply (r : Bare_Reply) ...

Definition PBFTdata_sender (m : data) : option name := match m with
  | PBFTdata_request (bare_request o t c) ⇒ Some (PBFTclient c)
  | PBFTdata_prepare (bare_prepare v n d i) ⇒ Some (PBFTreplica i)
  | PBFTdata_reply (bare_reply v t c i r) ⇒ Some (PBFTreplica i) ...

Definition PBFTget_contained_auth_data (m : msg) : list auth_data := match m with
  | REQUEST (request b a) ⇒ [(PBFTdata_request b,a)]
  | PREPARE (prepare b a) ⇒ [(PBFTdata_prepare b,a)]
  | REPLY (reply b a) ⇒ [(PBFTdata_reply b,a)] ...
```

3.4 Event Orderings

A typical way to reason about a distributed system is to reason about its possible runs, which are sometimes modeled as execution traces [72], and which are captured in LoE using *event orderings*. An *event ordering* is an abstract representation of a run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by system designers (see for example Fig. 1). As opposed to [72], a trace here is not just one sequence of events but instead can be seen as a collection of local traces (one local trace per sequential process), where a local trace is a collection of events all happening at the same location and ordered in time, and such that some events of different local traces are causally ordered. Event orderings are never instantiated. Instead, we express system properties as predicates on event orderings. A system satisfies such a property if every possible execution of the system satisfies the predicate. We first formally define the components of an event ordering, and then present the axioms that these components have to satisfy.

Components. An event ordering is formally defined as the tuple:[7]

```
Class EventOrdering :=
  { Event : Type;                          happenedBefore : Event → Event → Prop;
    loc : Event → name;                    direct_pred : Event → option Event;
    trigger : Event → option msg;          keys : Event → local_keys; }
```

where (1) Event is an abstract type of events; (2) happenedBefore is an ordering relation on events; (3) loc returns the location at which events happen; (4) direct_pred returns the direct local predecessor of an event when one exists, i.e., for all events except initial events; (5) given an event e, trigger either returns the message that triggered e, or it returns None to indicate that no information is available regarding the action that triggered the event (see below); (6) keys returns the keys a node can use at a given event to communicate with other nodes. The event orderings presented here are similar to the ones used in [3,71], which we adapted to handle Byzantine faults by modifying the type of trigger so that events can be triggered by arbitrary actions and not necessarily by the receipt of a message, and by adding support for authentication through keys.

The trigger function returns None to capture the fact that nodes can sometimes behave arbitrarily. This includes processes behaving correctly, i.e., according to their specifications; as well as (possibly malicious) processes deviating from their specifications. Note that this does not preclude from capturing the behavior of correct processes because for all event orderings where trigger returns None for an event where the node behaved correctly, there is a similar event ordering, where trigger returns the triggering message at that event. To model that at most f nodes out of n can be faulty we use the exists_at_most_f_faulty assumption, which enforces that trigger returns None at most f nodes.

Moreover, even though non-syntactically valid messages do not trigger events because they are discarded by message boxes, a triggering message could be

[7] A Coq type class is essentially a dependent record.

syntactically valid, but have an invalid signature. Therefore, it is up to the programmer to ensure that processes only react to messages with valid signatures using the verify function. Our authenticated_messages_were_sent_non_byz and exists_at_most_f_faulty assumptions presented in Sect. 3.6 are there to constrain trigger to ensure that at most f nodes out of n can diverge from their specifications, for example, by producing valid signatures even though they are not the nodes they claim to be (using leaked keys of other nodes).

Axioms. The following axioms characterize the behavior of these components:

1. Equality between events is decidable. Events are abstract entities that correspond to points in space/time that can be seen as pairs of numbers (one for the space coordinate and one for the time coordinate), for which equality is decidable.
2. The happened before relation is transitive and well-founded. This allows us to prove properties by induction on causal time. We assume here that it is not possible to infinitely go back in time, i.e., that there is a beginning of (causal) time, typically corresponding to the time a system started.
3. The direct predecessor e_2 of e_1 happens at the same location and before e_1. This makes local orderings sub-orderings of the happenedBefore ordering.
4. If an event e does not have a direct predecessor (i.e., e is an initial event) then there is no event happening locally before e.
5. The direct predecessor function is injective, i.e., two different events cannot have the same direct predecessor.
6. If an event e_1 happens locally before e_2 and e is the direct predecessor of e_2, then either $e = e_1$ or e_1 happens before e. From this, it follows that the direct predecessor function can give us the complete local history of an event.

Notation. We use $a \prec b$ to stand for (happenedBefore a b); $a \preceq b$ to stand for ($a \prec b$ or $a{=}b$); and $a \sqsubseteq b$ to stand for ($a \preceq b$ and loc a=loc b). We also sometimes write EO instead of EventOrdering.

Some functions take an event ordering as a parameter. For readability, we sometimes omit those when they can be inferred from the context. Similarly, we will often omit type declarations of the form (T : Type).

Correct Behavior. To prove properties about distributed systems, one only reasons about processes that have a correct behavior. To do so we only reason about events in event orderings that are correct in the sense that they were triggered by some message:

```
Definition isCorrect (e : Event) := match trigger e with Some m ⇒ True | None ⇒ False end.
Definition arbitrary (e : Event) := ∼ isCorrect e.
```

Next, we characterize correct replica histories as follows: (1) First we say that an event e has a correct trace if all local events prior to e are correct. (2) Then, we say that a node i has a correct trace before some event e, not necessarily happening at i, if all events happening before e at i have a correct trace:

```
Definition has_correct_bounded_trace (e : Event) := forall e', e' ⊑ e → isCorrect e'.
Definition has_correct_trace_before (e : Event) (i : name) :=
  forall e', e' ≺ e → loc e' = i → has_correct_bounded_trace e'.
```

3.5 Computational Model

Model. We now present our computational model, which we use when extracting OCaml programs. Unlike in EventML [71] where systems are first specified as *event observers* (abstract processes), and then later refined to executable code, we skip here event observers, and directly specify systems using executable state machines, which essentially consist of an update function and a current state. We define a system of distributed state machines as a function that maps names to state machines. Systems are parametrized by a function that associates state types with names in order to allow for different nodes to run different machines.

```
Definition Update S I O := S → I → (option S * O).
Record StateMachine S I O := MkSM { halted : bool; update : Update S I O; state : S }.
Definition System (F : name → Type) I O := forall (i : name), StateMachine (F i) I O.
```

where S is the type of the machine's state, I/O are the input/output types, and halted indicates whether the state machine is still running or not.

Let us now discuss how we relate state machines and events. We define state_sm_before_event and state_sm_after_event that compute a machine's state before and after a given event e. These states are computed by extracting the local history of events up to e using direct_pred, and then updating the state machine by running it on the triggering messages of those events. These functions return None if some arbitrary event occurs or the machine halts sometime along the way. Otherwise they return Some s, where s is the state of the machine updated according to the events. Therefore, assuming they return Some amounts to assuming that all events prior to e are correct, i.e., we can prove that if state_sm_after_event sm e = Some s then has_correct_trace_before e (loc e). As illustrated below, we use these functions to adopt a Hoare-like reasoning style by stating pre/post-conditions on the state of a process prior and after some event.

PBFT. We implement PBFT replicas as state machines, which we derive from an update function that dispatches input messages to the corresponding handlers. Finally, we define PBFTsys as the function that associates PBFTsm with replicas and a halted machine with clients (because we do not reason here about clients).

```
Definition PBFTupdate (i : Rep) := fun state msg ⇒ match msg with
  | REQUEST r ⇒ PBFThandle_request i state r
  | PREPARE p ⇒ PBFThandle_prepare i state p ...
Definition PBFTsm (i : Rep) := MkSM false (PBFTupdate i) (initial_state i).
Definition PBFTsys := fun name ⇒ match name with
  | PBFTreplica i ⇒ PBFTsm i | PBFTclient c ⇒ haltedSM end.
```

Let us illustrate how we reason about state machines through a simple example that shows that they maintain a view that only increases over time. It shows a local property, while Sect. 5 presents the distributed agreement property that makes use of the assumptions presented in Sect. 3.6. As mentioned above we prove such properties for all possible event orderings, which means that they are true for all possible runs of the system. In this lemma, *s1* is the state prior to the event *e*, and *s2* is the state after handling *e*. It does not have pre-conditions, and its post-condition states that the view in *s1* is smaller than the view in *s2*.

```
Lemma current_view_increases : forall (eo : EO) (e : Event) i s1 s2,
    state_sm_before_event (PBFTsm i) e = Some s1
    → state_sm_after_event (PPBFTsm i) e = Some s2
    → current_view s1 ≤ current_view s2.
```

3.6 Assumptions

Model. Let us now turn to the assumptions we make regarding the network and the behavior of correct and faulty nodes.

Assumption 1. Proving safety properties of crash fault-tolerant protocols that only require reasoning about past events, such as agreement, does not require reasoning about faults and faulty replicas. To prove such properties, one merely has to follow the causal chains of events back in time, and if a message is received by a node then it must have been sent by some node that had not crashed at that time. The state of affairs is different when dealing with Byzantine faults.

One issue it that Byzantine nodes can deviate from their specifications or impersonate other nodes. However, BFT protocols are designed in such a way that nodes only react to collections of messages, called *certificates*, that are larger than the number of faults. This means that there is always at least one correct node that can be used to track down causal chains of events.

A second issue is that, in general, we cannot assume that some received message was sent as such by the designated (correct) sender of the message because messages can be manipulated while in flight. As captured by the authenticated_messages_were_sent_or_byz predicate defined below,[8] we can only assume that the authenticated parts of the received message were actually sent by the designated senders, possibly inside larger messages, provided the senders did not leak their keys. As usual, we assume that attackers cannot break the cryptographic primitives, i.e., that they cannot authenticate messages without the proper keys [14].

```
1.Definition authenticated_messages_were_sent_or_byz (P : AbsProcess) :=
2. forall e (a : auth_data),
3.   In a (bind_op_list get_contained_auth_data (trigger e))
4.   → verify_auth_data (loc e) a (keys e) = true
```

[8] For readability, we show a slightly simplified version of this axiom. The full axiom can be found in https://github.com/vrahli/Velisarios/blob/master/model/EventOrdering.v.

```
5.  → exists e', e' ≺ e ∧ am_auth a = authenticate (am_data a) (keys e')
6.    ∧ ( (exists dst m,
7.           In a (get_contained_auth_data m) ∧ In (m,dst) (P eo e')
8.           ∧ data_sender (loc e) (am_data a) = Some (loc e'))
9.         ∨
10.        (exists e",
11.          e" ⪯ e' ∧ arbitrary e' ∧ arbitrary e" ∧ got_key_for (loc e) (keys e") (keys e')
12.          ∧ data_sender (loc e) (am_data a) = Some (loc e")) ).
```

This assumption says that if the authenticated piece of data a is part of the message that triggered some event e (L.3), and a is verified (L.4), then there exists a prior event e' such that the data was authenticated while handling e' using the keys available at that time (L.5). Moreover, (1) either the sender of the data was correct while handling e' and sent the data as part of a message following the process described by P (L.6–8); or (2) the node at which e' occurred was Byzantine at that time, and either it generated the data itself (e.g. when $e"$=e'), or it impersonated some other replica (by obtaining the keys that some node leaked at event $e"$) (L.10–12).

We used a few undefined abstractions in this predicate: An AbsProcess is an abstraction of a process, i.e., a function that returns the collection of messages generated while handling a given event: (`forall` (*eo* : EO) (*e* : Event), list DirectedMsg). The bind_op_list function is wrapped around get_contained_auth_data to handle the fact that trigger might return None, in which case bind_op_list returns nil. The verify_auth_data function takes an authenticated message a and some keys and: (1) invokes data_sender (defined in Sect. 3.3) to extract the expected sender s of a; (2) searches among its keys for a receiving_key that it can use to verify that s indeed authenticated a; and (3) finally verifies the authenticity of a using that key and the verify function. The authenticate function simply calls create and uses the sending keys to create tokens. The got_key_for function takes a name i and two local_keys *lk1* and *lk2*, and states that the sending keys for i in *lk1* are all included in *lk2*.

However, it turns out that because we never reason about faulty nodes, we never have to deal with the right disjunct of the above formula. Therefore, this assumption about received messages can be greatly simplified when we know that the sender is a correct replica, which is always the case when we use this assumption because BFT protocols as designed so that there is always a correct node that can be used to track down causal chains of events. We now define the following simpler assumption, which we have proved to be a consequence of authenticated_messages_were_sent_or_byz:

```
Definition authenticated_messages_were_sent_non_byz (P : AbsProcess) :=
  forall (e : Event) (a : auth_data) (c : name),
    In a (bind_op_list get_contained_auth_data (trigger e))
    → has_correct_trace_before e c
    → verify_auth_data (loc e) a (keys e) = true
    → data_sender (loc e) (am_data a) = Some c
    → exists e' dst m, e' ≺ e ∧ loc e' = c.
        ∧ am_auth a = authenticate (am_data a) (keys e')
        ∧ In a (get_contained_auth_data m)
        ∧ In (m,dst) (P eo e')
```

As opposed to the previous formula, this one assumes that the authenticated data was sent by a correct replica, which has a correct trace prior to the event e—the event when the message containing a was handled.

Assumption 2. Because processes need to store their keys to sign and verify messages, we must connect those keys to the ones in the model. We do this through the correct_keys assumption, which states that for each event e, if a process has a correct trace up to e, then the keys (keys e) from the model are the same as the ones stored in its state (which are computed using state_sm_before_event).

Assumption 3. Finally, we present our assumption regarding the number of faulty nodes. There are several ways to state that there can be at most f faulty nodes. One simple definition is (where node is a subset of name as discussed in Sect. 4.2):

```
Definition exists_at_most_f_faulty (E : list Event) (f : nat) :=
  exists (faulty : list node), length faulty ≤ f
    ∧ forall e1 e2, In e2 E → e1 ≼ e2 → ∼ In (loc e1) faulty
          → has_correct_bounded_trace e1.
```

This assumption says that at most f nodes can be faulty by stating that the events happening at nodes that are not in the list of faulty nodes *faulty*, of length f, are correct up to some point characterized by the partial cut E of a given event ordering (i.e., the collection of events happening before those in E).

PBFT Assumption 4. In addition to the ones above, we made further assumptions about PBFT. Replicas sometimes send message hashes instead of sending the entire messages. For example, pre-prepare messages contain client requests, but prepare and commit messages simply contain digests of client requests. Consequently, our PBFT formalization is parametrized by the following *create* and *verify* functions, and we assume that the create function is collision resistant:[9]

```
Class PBFThash := MkPBFThash {
  create_hash : list PBFTmsg → digest; verify_hash : list PBFTmsg → digest → bool; }.
Class PBFThash_axioms := MkPBFThash_axioms {
  create_hash_collision_resistant :
    forall msgs1 msgs2, create_hash msgs1 = create_hash msgs2 → msgs1 = msgs2; }.
```

The version of PBFT, called PBFT-PK in [14], that we implemented relies on digital signatures. However, we did not have to make any more assumptions regarding the cryptographic primitives than the ones presented above, and in particular we did not assume anything that is true about digital signatures and false about MACs. Therefore, our safety proof works when using either digital signatures or MAC vectors. As discussed below, this is true because we adapted the way messages are verified (we have not verified the MAC version of PBFT but a slight variant of PBFT-PK) and because we do not deal with liveness.

[9] Note that our current collision resistant assumption is too strong because it is always possible to find two distinct messages that are hashed to the same hash. We leave it to future work to turn it into a more realistic probabilistic assumption.

As Castro showed [14, Chap. 3], PBFT-PK has to be adapted when digital signatures are replaced by MAC vectors. Among other things, it requires "significant and subtle changes to the view change protocol" [14, Sect. 3.2]. Also, to the best of our knowledge, in PBFT-PK backups do not check the authenticity of requests upon receipt of pre-prepares. They only check the authenticity of requests before executing them [14, p. 42]. This works when using digital signatures but not when using MACs: one backup might not execute the request because its part of the MAC vector does not check out, while another backup executes the request because its part of the MAC vector checks out, which would lead to inconsistent states and break safety. Castro lists other problems related to liveness.

Instead, as in the MAC version of PBFT [14, p. 42], in our implementation we always check requests' validity when checking the validity of a pre-prepare. If we were to check the validity of requests only before executing them, we would have to assume that two correct replicas would either both be able to verify the data, or both would not be able to do so. This assumption holds for digital signatures but not for MAC vectors.

4 Methodology

Because distributed systems are all about exchanging information among nodes, we have developed a theory that captures abstractions and reasoning patterns to deal with knowledge dissemination (see Sect. 4.4). In the presence of faulty nodes, one has to ensure that this knowledge is reliable. Fault-tolerant state-machine replication protocols provide such guarantees by relying on certificates, which ensure that we can always get hold of a correct node to trace back information through the system. This requires reasoning about the past, i.e., reasoning by induction on causal time using the happenedBefore relation.

4.1 Automated Inductive Reasoning

We use induction on causal time to prove both distributed and local properties. As discussed here, we automated the typical reasoning pattern we use to prove local properties. As an example, in our PBFT formalization, we proved the following local property: if a replica has a prepare message in its log, then it either received or generated it. Moreover, as for any kinds of programs, using Velisarios we prove local properties about processes by reasoning about all possible paths they can take when reacting upon messages. Thus, a typical proof of such a lemma using Velisarios goes as follows: (1) we go by induction on events; (2) we split the code of a process into all possible execution paths; (3) we prune the paths that could not happen because they invalidate some hypotheses of the lemma being proved; and (4) we automatically prove some other cases by induction hypothesis. We packaged this reasoning as a Coq tactic, which in practice can significantly reduce the number of cases to prove, and used this automation

technique to prove local properties of PBFT, such as Castro's A.1.2 local invariants [14]. Because of PBFT's complexity, our Coq tactic typically reduces the number of cases to prove from between 50 to 60 cases down to around 7 cases, sometimes less, as we show in this histogram of goals left to interactively prove after automation:

# of goals left to prove	0	1	2	3	4	5	6	7	8
# of lemmas	8	1	5	4	4	2	9	17	3

4.2 Quorums

As usual, we use quorum theory to trace back correct information between nodes. A (Byzantine) quorum w.r.t. a given set of nodes N, is a subset Q of N, such that $f + 1 \leq (2 * |Q|) - |N|$ (where $|X|$ is the size of X), i.e. every two quorums intersect [59,83] in sufficiently many replicas.[10] Typically, a quorum corresponds to a majority of nodes that agree on some property. In case of state machine replication, quorums are used to ensure that a majority of nodes agree to update the state using the same operation. If we know that two quorums intersect, then we know that both quorums agree, and therefore that the states cannot diverge. In order to reason about quorums, we have proved the following general lemma:[11]

```
Lemma overlapping_quorums :
  forall (l1 l2 : NRlist node), exists Correct,
    (length l1 + length l2) - num_nodes ≤ length Correct
    ∧ subset Correct l1 ∧ subset Correct l2 ∧ no_repeats Correct.
```

This lemma implies that if we have two sets of nodes *l1* and *l2* (NRlist ensures that the sets have no repeats), such that the sum of their length is greater than the total number of nodes (num_nodes), there must exist an overlapping subset of nodes (*Correct*). We use this result below in Sect. 4.4.

The node type parameter is the collection of nodes that can participate in quorums. For example, PBFT replicas can participate in quorums but clients cannot. This type comes with a node2name function to convert nodes into names.

4.3 Certificates

Lemmas that require reasoning about several replicas are much more complex than local properties. They typically require reasoning about some information computed by a collection of replicas (such as quorums) that vouch for the information. In PBFT, a collection of $2f + 1$ messages from different replicas is called

[10] We use here Castro's notation where quorums are *majority* quorums [79] (also called *write quorums*) that require intersections to be non-empty, as opposed to *read quorums* that are only required to intersect with write quorums [36].

[11] We present here a simplified version for readability.

a *strong (or quorum) certificate*, and a collection of $f+1$ messages from different replicas is called a *weak certificate*.

When working with strong certificates, one typically reasons as follows: (1) Because PBFT requires $3f+1$ replicas, two certificates of size $2f+1$ always intersect in $f+1$ replicas. (2) One message among those $f+1$ messages must be from a correct replica because at most f replicas can be faulty. (3) This correct replica can vouch for the information of both quorums—we use that replica to trace back the corresponding information to the point in space/time where/when it was generated. We will get back to this in Sect. 4.4.

When working with weak certificates, one typically reasons as follows: Because, the certificate has size $f+1$ and there are at most f faulty nodes, there must be one correct replica that can vouch for the information of the certificate.

4.4 Knowledge Theory

Model. Let us now present an excerpt of our distributed epistemic knowledge library. Knowledge is a widely studied concept [10,30,31,37–39,70]. It is often captured using possible-worlds models, which rely on Kripke structures: an agent knows a fact if that fact is true in all possible worlds. For distributed systems, agents are nodes and a possible world at a given node is essentially one that has the same local history as the one of the current world, i.e., it captures the current state of the node. As Halpern stresses, e.g. in [37], such a definition of knowledge is *external* in the sense that it cannot necessarily be computed, though some work has been done towards deriving programs from knowledge-based specifications [10]. We follow a different, more pragmatic and computational approach, and say that a node knows some piece of data if it is stored locally, as opposed to the external and logical notion of knowing facts mentioned above. This computational notion of knowledge relies on exchanging messages to propagate it, which is what is required to derive programs from knowledge-based specifications (i.e., to compute that some knowledge is gained [20,37]).

We now extend the model presented in Sect. 3 with two epistemic modal operators *know* and *learn* that express what it means for a process to know and learn some information, and which bear some resemblance with the *fact discovery* and *fact publication* notions discussed in [38]. Formally, we extend our model with the following parameters, which can be instantiated as many times as needed for all the pieces of known/learned data that one wants to reason about—see below for examples:

```
Class LearnAndKnow := MkLearnAndKnow {
    lak_data : Type;          lak_data2info : lak_data → lak_info;
    lak_info : Type;          lak_know : lak_data → lak_memory → Prop;
    lak_memory : Type;        lak_data2owner : lak_data → node;
                              lak_data2auth : lak_data → auth_data; }.
```

The lak_data type is the type of "raw" data that we have knowledge of; while lak_info is some distinct information that might be shared by different pieces

of data. For example, PBFT replicas collect batches of $2f + 1$ (pre-)prepare messages from different replicas, that share the same view, sequence number, and digest. In that case, the (pre-)prepare messages are the raw data that contain the common information consisting of a view, a sequence number, and a digest. The lak_memory type is the type of objects used to store one's knowledge, such as a state machine state. One has to provide a lak_data2info function to extract the information embedded in some piece of data. The lak_know predicate explains what it means to know some piece of data. The lak_data2owner function extracts the "owner" of some piece of data, typically the node that generated the data. In order to authenticate pieces of data, the lak_data2auth function extracts some piece of authenticated data from some piece of raw data. For convenience, we define the following wrapper around lak_data2owner:

```
Definition lak_data2node (d : lak_data) : name := node2name (lak_data2owner d).
```

Let us now turn to the two main components of our theory, namely the know and learn epistemic modal operators. These operators provide an abstraction barrier: they allow us to abstract away from *how* knowledge is stored and computed, in order to focus on the mere *fact* that we have that knowledge.

```
Definition know (sm : node → StateMachine lak_memory) (e : Event) (d : lak_data) :=
  exists mem i, loc e = node2name i
    ∧ state_sm_after_event (sm i) e = Some mem
    ∧ lak_know d mem.
```

where we simply write (StateMachine S) for a state machine with a state of type S, that takes messages as inputs, and outputs lists of directed messages. This states that the state machine (sm i) knows the data d at event e if its state is mem at e and (lak_know d mem) is true. We define learn as follows:

```
Definition learn (e : Event) (d : lak_data) :=
  exists i, loc e = node2name i
    ∧ In (lak_data2auth d) (bind_op_list get_contained_auth_data (trigger e))
    ∧ verify_auth_data (loc e) (lak_data2auth d) (keys e) = true.
```

This states that a node learns d at some event e, if e was triggered by a message that contains the data d. Moreover, because we deal with Byzantine faults, we require that to learn some data one has to be able to verify its authenticity.

Next, we define a few predicates that are useful to track down knowledge. The first one is a local predicate that says that for a state machine to know about a piece of information it has to either have learned it or generated it.

```
Definition learn_or_know (sm : node → StateMachine lak_memory) :=
  forall (d : lak_data) (e : Event),
    know sm e d → (exists e', e' ⊑ e ∧ learn e' d) ∨ lak_data2node d = loc e.
```

The next one is a distributed predicate that states that if one learns some piece of information that is owned by a correct node, then that correct node must have known that piece of information:

```
Definition learn_if_know (sm : node → StateMachine lak_memory) :=
  forall (d : lak_data) (e : Event),
    (learn e d ∧ has_correct_trace_before e (lak_data2node d))
    → exists e', e' ≺ e ∧ loc e' = lak_data2node d ∧ know sm e' d.
```

Using these two predicates, we have proved this general lemma about knowledge propagating through nodes:

```
Lemma know_propagates :
  forall (e : Event) (sm : node → StateMachine lak_memory) (d : lak_data),
    (learn_or_know sm ∧ learn_if_know sm)
    → (know sm e d ∧ has_correct_trace_before e (lak_data2node d))
    → exists e', e' ⪯ e ∧ loc e' = lak_data2node d ∧ know sm e' d.
```

This lemma says that, assuming learn_or_know and learn_if_know, if one knows at some event e some data d that is owned by a correct node, then that correct node must have known that data at a prior event e'. We use this lemma to track down information through correct nodes.

As mentioned in Sect. 4.3, when reasoning about distributed systems, one often needs to reason about certificates, i.e., about collections of messages from different sources. In order to capture this, we introduce the following know_certificate predicate, which says that the state machine sm knows the information i at event e if there exists a list l of pieces of data of length at least k (the certificate size) that come from different sources, and such that sm knows each of these pieces of data, and each piece of data carries the common information nfo:

```
Definition know_certificate (sm : node → StateMachine lak_memory)
           (e : Event) (k : nat) (nfo : lak_info) (P : list lak_data → Prop) :=
  exists (l : list lak_data),
    k ≤ length l ∧ no_repeats (map lak_data2owner l) ∧ P l
    ∧ forall d, In d l → (know sm e d ∧ nfo = lak_data2info d).
```

Using this predicate, we can then combine the quorum and knowledge theories to prove the following lemma, which captures the fact that if there are two quorums for information $nfo1$ (known at $e1$) and $nfo2$ (known at $e2$), and the intersection of the two quorums is guaranteed to contain a correct node, then there must be a correct node (at which $e1'$ and $e2'$ happen) that owns and knows both $nfo1$ and $nfo2$—this lemma follows from know_propagates and overlapping_quorums:

```
Lemma know_in_intersection :
  forall (sm : node → StateMachine lak_memory) (e1 e2 : Event) (nfo1 nfo2 : lak_info)
         (k f : nat) (P : list lak_data → Prop) (E : list Event),
    (learn_or_know sm ∧ learn_if_know sm)
    → (k ≤ num_nodes ∧ num_nodes + f < 2 * k)
    → (exists_at_most_f_faulty E f ∧ In e1 E ∧ In e2 E)
    → (know_certificate sm e1 k nfo1 P ∧ know_certificate sm e2 k nfo2 P)
    → exists e1' e2' d1 d2, loc e1' = loc e2' ∧ e1' ⪯ e1 ∧ e2' ⪯ e2
        ∧ loc e1' = lak_data2node d1 ∧ loc e2' = lak_data2node d2
        ∧ know sm e1' d1 ∧ know sm e2' d2
        ∧ i1 = lak_data2info d1 ∧ i2 = lak_data2info d2.
```

Similarly, we proved the following lemma, which captures the fact that there is always a correct replica that can vouch for the information of a weak certificate:

```
Lemma know_weak_certificate :
 forall (e : Event) (k f : nat) (nfo : lak_info) (P : list lak_data → Prop) (E : list Event),
  (f < k ∧ exists_at_most_f_faulty E f ∧ In e E ∧ know_certificate e k nfo P)
  → exists d, has_correct_trace_before e (node2node d) ∧ know e d ∧ nfo = lak_data2info d.
```

PBFT. One of the key lemmas to prove PBFT's safety says that if two correct replicas have prepared some requests with the same sequence and view numbers, then the requests must be the same [14, Inv.A.1.4]. As mentioned in Sect. 2.1, a replica has prepared a request if it received pre-prepare and prepare messages from a quorum of replicas. To prove this lemma, we instantiated LearnAndKnow as follows: lak_data can either be a pre-prepare or a prepare message; lak_info is the type of triples view/sequence number/digest; lak_memory is the type of states maintained by replicas; lak_data2info extracts the view, sequence number and digest contained in pre-prepare and prepare messages; lak_know states that the pre-prepare or prepare message is stored in the state; lak_data2owner extracts the sender of the message; and lak_data2auth is similar to the PBFTget_contained_auth_data function presented in Sect. 3.6. The two predicates learn_or_know and learn_if_know, which we proved using the tactic discussed in Sect. 4.1, are true about this instance of LearnAndKnow. Inv.A.1.4 is then a straightforward consequence of know_in_intersection applied to the two quorums.

5 Verification of PBFT

Agreement. Velisarios is designed as a general, reusable, and extensible framework that can be instantiated to prove the correctness of any BFT protocol. We demonstrated its usability by proving that our PBFT implementation satisfies the standard agreement property, which is the crux of linearizability (we leave linearizability for future work—see Sect. 2.2 for a high-level definition). Agreement states that, regardless of the view, any two replies sent by correct replicas *i1* and *i2* at events *e1* and *e2* for the same timestamp *ts* to the same client *c* contain the same replies. We proved that this is true in any event ordering that satisfies the assumptions from Sect. 3.6:[12]

```
Lemma agreement :
  forall (eo : EventOrdering) (e1 e2 : Event) (v1 v2 : View) (ts : Timestamp)
         (c : Client) (i1 i2 : Rep) (r1 r2 : Request) (a1 a2 : list Token),
  authenticated_messages_were_sent_or_byz_sys eo PBFTsys ∧ correct_keys eo
  → (exists_at_most_f_faulty [e1,e2] f ∧ loc e1 = PBFTreplica i1 ∧ loc e2 = PBFTreplica i2)
  → In (send_reply v1 ts c i1 r1 a1) (output_system_on_event PBFTsys e1)
  → In (send_reply v2 ts c i2 r2 a2) (output_system_on_event PBFTsys e2)
  → r1 = r2.
```

[12] See agreement in https://github.com/vrahli/Velisarios/blob/master/PBFT/PBFTagreement.v.

where Timestamps are nats; authenticated_messages_were_sent_or_byz_sys is defined on systems using authenticated_messages_were_sent_or_byz; the function output_system_on_event is similar to state_sm_after_event (see Sect. 3.5) but returns the outputs of a given state machine at a given event instead of returning its state; and send_reply builds a reply message. To prove this lemma, we proved most of the invariants stated by Castro in [14, Appendix A]. In addition, we proved that if the last executed sequence number of two correct replicas is the same, then these two replicas have, among other things, the same service state.[13]

As mentioned above, because our model is based on LoE, we only ever prove such properties by induction on causal time. Similarly, Castro proved most of his invariants by induction on the length of the executions. However, he used other induction principles to prove some lemmas, such as Inv.A.1.9, which he proved by induction on views [14, p. 151]. This invariant says that prepared requests have to be consistent with the requests sent in pre-prepare messages by the primary. A straightforward induction on causal time was more natural in our setting.

Castro used a simulation method to prove PBFT's safety: he first proved the safety of a version without garbage collection and then proved that the version with garbage collection implements the one without. This requires defining two versions of the protocol. Instead, we directly prove the safety of the one with garbage collection. This involved proving further invariants about stored, received and sent messages, essentially that they are always within the water marks.

Proof Effort. In terms of proof effort, developing Velisarios and verifying PBFT's agreement property took us around 1 person year. Our generic Velisarios framework consists of around 4000 lines of specifications and around 4000 lines of proofs. Our verified implementation of PBFT consists of around 20000 lines of specifications and around 22000 lines of proofs.

6 Extraction and Evaluation

Extraction. To evaluate our PBFT implementation (i.e., PBFTsys defined in Sect. 3.5—a collection of state machines), we generate OCaml code using Coq's extraction mechanism. Most parameters, such as the number of tolerated faults, are instantiated before extraction. Note that not all parameters need to be instantiated. For example, as mentioned in Sect. 3.1, neither do we instantiate event orderings, nor do we instantiate our assumptions (such as exists_at_most_f_faulty), because they are not used in the code but are only used to prove that properties are true about all possible runs. Also, keys, signatures, and digests are only instantiated by stubs in Coq. We replace those stubs when extracting OCaml code by implementations provided by the nocrypto [66] library, which is the cryptographic library we use to hash, sign, and verify messages (we use RSA).

[13] See same_states_if_same_next_to_execute in https://github.com/vrahli/Velisarios/blob/master/PBFT/PBFTsame_states.v.

Evaluation. To run the extracted code in a real distributed environment, we implemented a small trusted runtime environment in OCaml that uses the Async library [5] to handle sender/receiver threads. We show among other things here that the average latency of our implementation is acceptable compared to the state of the art BFT-SMaRt [8] library. Note that because we do not offer a new protocol, but essentially a re-implementation of PBFT, we expect that on average the scale will be similar in other execution scenarios such as the ones studied by Castro in [14]. We ran our experiments using desktops with 16 GB of memory, and 8 i7-6700 cores running at 3.40 GHz. We report some of our experiments where we used a single client, and a simple state machine where the state is a number, and an operation is either adding or subtracting some value.

We ran a local simulation to measure the performance of our PBFT implementation without network and signatures: when 1 client sends 1 million requests, it takes on average 27.6 µs for the client to receive $f + 1$ ($f = 1$) replies.

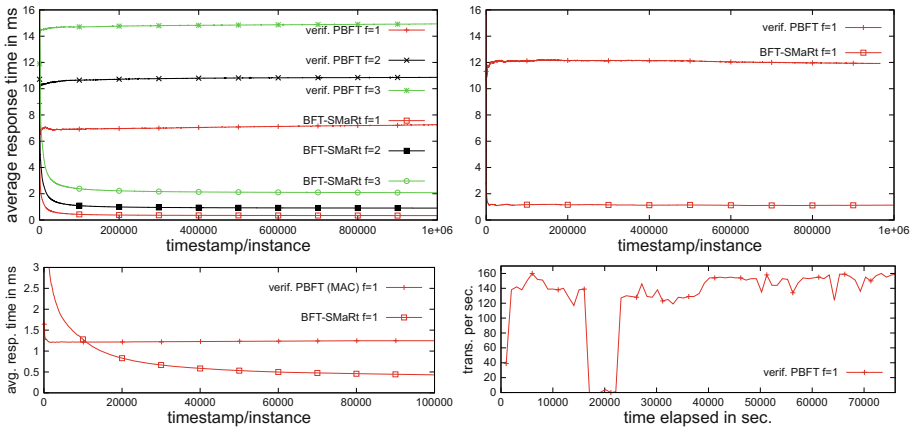

Fig. 3. (1) Single machine (top/left); (2) several machines (top/right); (3) single machine using MACs (bottom/left); (4) view change response time (bottom/right)

Top/left of Fig. 3 shows the experiment where we varied f from 1 to 3, and replicas sent messages, signed using RSA, through sockets, but on a single machine. As mentioned above, we implemented the digital signature-based version of PBFT, while BFT-SMaRt uses a more efficient MAC-based authentication scheme, which in part explains why BFT-SMaRt is around one order of magnitude faster than our implementation. As in [14, Table 8.9], we expect a similar improvement when using the more involved, and as of yet not formally verified, MAC-based version of PBFT (bottom/left of Fig. 3 shows the average response time when replacing digital signatures by MACs, without adapting the rest of the protocol). Top/right of Fig. 3 presents results when running our

version of PBFT and BFT-SMaRt on several machines, for $f = 1$. Finally, bottom/right of Fig. 3 shows the response time of our view-change protocol. In this experiment, we killed the primary after 16 s of execution, and it took around 7 s for the system to recover.

Trusted Computing Base. The TCB of our system includes: (1) the fact that our LoE model faithfully reflects the behavior of distributed systems (see Sect. 3.4); (2) the validity of our assumptions: authenticated_messages_were_sent_or_byz; exists_at_most_f_faulty; correct_keys; and create_hash_collision_resistant (Sect. 3.6); (3) Coq's logic and implementation; (4) OCaml and the nocrypto and Async libraries we use in our runtime environment, and the runtime environment itself (Sect. 6); (5) the hardware and software on which our framework is running.

7 Related Work

Our framework is not the first one for implementing and reasoning about the correctness of distributed systems (see Fig. 4). However, to the best of our knowledge, (1) it is the first theorem prover based tool for verifying the correctness of asynchronous Byzantine fault-tolerant protocols and their implementations; and (2) we provide the first mechanical proof of the safety of a PBFT implementation. Velisarios has evolved from our earlier EventML framework [71], primarily to reason about Byzantine faults and distributed epistemic knowledge.

	Running code	Byz. (synch.)	Byz. (asynch.)
IronFleet/EventML/Verdi/Disel/PSync	✓	✗	✗
HO-model/PVS	✗	✓	✗
Event-B	✓/✗	✓	✗
IOA/TLA$^+$/ByMC	✗	✓	✓
Velisarios	✓	✓	✓

Fig. 4. Comparison with related work

7.1 Logics and Models

IOA [33–35,78] is the model used by Castro [14] to prove PBFT's safety. It is a programming/specification language for describing asynchronous distributed systems as I/O automata [58] (labeled state transition systems) and stating their properties. While IOA is state-based, the logic we use in this paper is event-based. IOA can interact with a large range of tools such as type checkers, simulators, model checkers, theorem provers, and there is support for synthesis of Java code [78]. In contrast, our methodology allows us to both implement and verify protocols within the same tool, namely Coq.

TLA$^+$ [24,51] is a language for specifying and reasoning about systems. It combines: (1) TLA [52], which is a temporal logic for describing systems [51], and (2) set theory, to specify data structures. TLAPS [24] uses a collection of theorem provers, proof assistants, SMT solvers, and decision procedures to mechanically check TLA proofs. Model checker integration helps catch errors before verification attempts. TLA$^+$ has been used in a large number of projects (e.g., [12,18,44,56,63,64]) including proofs of safety and liveness of Multi-Paxos [18], and safety of a variant of an abstract model of PBFT [13]. To the best of our knowledge, TLA$^+$ does not perform program synthesis.

The Heard-Of (HO) Model [23] requires processes to execute in lock-step through rounds into which the distributed algorithms are divided. Asynchronous fault-tolerant systems are treated as synchronous systems with adversarial environments that cause messages to be dropped. The HO-model was implemented in Isabelle/HOL [22] and used, for example, to verify the EIGByz [7] Byzantine agreement algorithm for synchronous systems with reliable links. This formalization uses the notion of *global state of the system* [19], while our approach relies on Lamport's *happened before* relation [53], which does not require reasoning about a distributed system as a single entity (a global state). Model checking and the HO-model were also used in [21,80,81] for verifying the crash fault-tolerant consensus algorithms presented in [23]. To the best of our knowledge, there is no tool that allows generating code from algorithms specified using the HO-model.

Event-B [1] is a set-theory-based language for modeling reactive systems and for *refining* high-level abstract specifications into low-level concrete ones. It supports code generation [32,61], with some limitations (not all features are covered). The Rodin [2] platform for Event-B provides support for refinement, and automated and interactive theorem proving. Both have been used in a number of projects, such as: to prove the safety and liveness of self-$\star$ systems [4]; to prove the agreement and validity properties of the synchronous crash-tolerant Floodset consensus algorithm [57]; and to prove the agreement and validity of synchronous Byzantine agreement algorithms [50]. In [50], the authors assume that messages cannot be forged (using PBFT, at most f nodes can forge messages), and do not verify implementations of these algorithms.

7.2 Tools

Verdi [85,86] is a framework to develop and reason about distributed systems using Coq. As in our framework, Verdi leaves no gaps between verified and running code. Instead, OCaml code is extracted directly from the verified Coq implementation. Verdi provides a compositional way of specifying distributed systems. This is done by applying *verified system transformers*. For example, Raft [67]—an alternative to Paxos—transforms a distributed system into a crash-tolerant one. One difference between our respective methods is that they verify a system by reasoning about the evolution of its global state, while we use Lamport's happened before relation. Moreover, they do not deal with the full spectrum of arbitrary faults (e.g., malicious faults).

Disel [75,84] is a verification framework that implements a separation-style program logic, and that enables compositional verification of distributed systems.

IronFleet [40,41] is a framework for building and reasoning about distributed systems using Dafny [55] and the Z3 theorem prover [62]. Because systems are both implemented in and verified using Dafny, IronFleet also prevents gaps between running and verified code. It uses a combination of TLA-style state-machine refinements [51] to reason about the distributed aspects of protocols, and Floyd-Hoare-style imperative verification techniques to reason about local behavior. The authors have implemented, among other things, the Paxos-based state machine replication library IronRSL, and verified its safety and liveness.

PSync [28] is a domain specific language embedded in Scala, that enables executing and verifying fault-tolerant distributed algorithms in synchronous and partially asynchronous networks. PSync is based on the HO-model, and has been used to implement several crash fault-tolerant algorithms. Similar to the Verdi framework, PSync makes use of a notion of global state and supports reasoning based on the multi-sorted first-order *Consensus verification logic* (CL) [27]. To prove safety, users have to provide invariants, which CL checks for validity. Unlike Verdi, IronFleet and PSync, we focus on Byzantine faults.

ByMC is a model checker for verifying safety and liveness of fault-tolerant distributed algorithms [47–49]. It applies an automated method for model checking parametrized threshold-guarded distributed algorithms (e.g., processes waiting for messages from a majority of distinct senders). ByMC is based on a short counter-example property, which says that if a distributed algorithm violates a temporal specification then there is a counterexample whose length is bounded and independent of the parameters (e.g. the number of tolerated faults).

Ivy [69] allows debugging infinite-state systems using bounded verification, and formally verifying their safety by gradually building universally quantified inductive invariants. To the best of our knowledge, Ivy does not support faults.

Actor Services [77] allows verifying the distributed and functional properties of programs communicating via asynchronous message passing at the level of the source code (they use a simple Java-like language). It supports modular reasoning and proving liveness. To the best of our knowledge, it does not deal with faults.

PVS has been extensively used for verification of synchronous systems that tolerate malicious faults such as in [74], to the extent that its design was influenced by these verification efforts [68].

8 Conclusions and Future Work

We introduced Velisarios, a framework to implement and reason about BFT-SMR protocols using the Coq theorem prover, and described a methodology based on learn/know epistemic modal operators. We used this framework to

prove the safety of a complex system, namely Castro's PBFT protocol. In the future, we plan to also tackle liveness/timeliness. Indeed, proving the safety of a distributed system is far from being enough: a protocol that does not run (which is not live) is useless. Following the same line of reasoning, we want to tackle timeliness because, for real world systems, it is not enough to prove that a system will *eventually reply*. One often desires that the system replies in a timely fashion.

References

1. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Anand, A., Knepper, R.: ROSCoq: robots powered by constructive reals. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 34–50. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_3
4. Andriamiarina, M.B., Méry, D., Singh, N.K.: Analysis of self-$\star$ and P2P systems using refinement. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 117–123. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_9
5. Async. https://janestreet.github.io/guide-async.html
6. Aublin, P.-L., Mokhtar, S.B., Qéuma, V.: RBFT: redundant Byzantine fault tolerance. In: ICDCS 2013, pp. 297–306. IEEE Computer Society (2013)
7. Bar-Noy, A., Dolev, D., Dwork, C., Raymond Strong, H.: Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. Inf. Comput. **97**(2), 205–233 (1992)
8. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: DSN 2014, pp. 355–362. IEEE (2014)
9. Bickford, M.: Component specification using event classes. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 140–155. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02414-6_9
10. Bickford, M., Constable, R.C., Halpern, J.Y., Petride, S.: Knowledge-based synthesis of distributed systems using event structures. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 449–465. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_30
11. Bickford, M., Constable, R.L., Rahli, V.: Logic of events, a framework to reason about distributed systems. In: Languages for Distributed Algorithms Workshop (2012)
12. Bolosky, W.J., Douceur, J.R., Howell, J.: The Farsite project: a retrospective. Oper. Syst. Rev. **41**(2), 17–26 (2007)
13. Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm. http://lamport.azurewebsites.net/tla/byzpaxos.html
14. Castro, M.: Practical Byzantine Fault Tolerance. Also as Technical report MIT-LCS-TR-817. Ph.D. MIT, January 2001
15. Castro, M., Liskov, B.: A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Technical Memo MIT-LCS-TM-590. MIT, June 1999

16. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: OSDI 1999, pp. 173–186. USENIX Association (1999)
17. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
18. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 119–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_8
19. Mani Chandy, K., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
20. Mani Chandy, K., Misra, J.: How processes learn. Distrib. Comput. **1**(1), 40–52 (1986)
21. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04420-5_10
22. Charron-Bost, B., Debrat, H., Merz, S.: Formal verification of consensus algorithms tolerating malicious faults. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 120–134. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3_11
23. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. Distrib. Comput. **22**(1), 49–71 (2009)
24. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^+$ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 142–148. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_12
25. The Coq Proof Assistant. http://coq.inria.fr/
26. Distler, T., Cachin, C., Kapitza, R.: Resource-efficient Byzantine fault tolerance. IEEE Trans. Comput. **65**(9), 2807–2819 (2016)
27. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 161–181. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_10
28. Dragoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL 2016, pp. 400–415. ACM (2016)
29. Dragoi, C., Henzinger, T.A., Zufferey, D.: The need for language support for fault-tolerant distributed systems. In: SNAPL 2015. LIPIcs, vol. 32, pp. 90–102. Schloss Dagstuhl - Leibniz-Zentrum fuerInformatik (2015)
30. Dwork, C., Moses, Y.: Knowledge and common knowledge in a Byzantine environment: crash failures. Inf. Comput. **88**(2), 156–186 (1990)
31. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Knowledge-based programs. Distrib. Comput. **10**(4), 199–225 (1997)
32. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_20
33. Garland, S., Lynch, N., Tauber, J., Vaziri, M.: IOA user guide and reference manual. Technical report MIT/LCS/TR-961. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (2004)

34. Garland, S.J., Lynch, N.: Using I/O automata for developing distributed systems. In: Foundations of Component Based Systems, pp. 285–312. Cambridge University Press, New York (2000)
35. Georgiou, C., Lynch, N., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the IOA language. Int. J. Softw. Tools Technol. Transf. **11**, 153–171 (2009)
36. Gifford, D.K.: Weighted voting for replicated data. In: SOSP 1979, pp. 150–162. ACM (1979)
37. Halpern, J.Y.: Using reasoning about knowledge to analyze distributed systems. Ann. Rev. Comput. Sci. **2**(1), 37–68 (1987). https://doi.org/10.1146/annurev.cs.02.060187.000345
38. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990)
39. Halpern, J.Y., Zuck, L.D.: A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. J. ACM **39**(3), 449–478 (1992)
40. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: SOSP 2015, pp. 1–17. ACM (2015)
41. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017)
42. Herlihy, M., Wing, J.M.: Axioms for concurrent objects. In: POPL 1987, pp. 13–26. ACM Press (1987)
43. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S.: Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats. Advances in Information Security, vol. 54. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-0977-9
44. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M.R., Yuan, Y.: Checking cache-coherence protocols with TLA$^+$. Formal Methods Syst. Des. **22**(2), 125–131 (2003)
45. Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S.V., Schröder-Preikschat, W., Stengel, K.: CheapBFT: resource-efficient Byzantine fault tolerance. In: EuroSys 2012, pp. 295–308. ACM (2012)
46. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing Bitcoin security and performance with strong consistency via collective signing. In: USENIX Security Symposium, pp. 279–296. USENIX Association (2016)
47. Konnov, I.V., Lazic, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL 2017, pp. 719–734. ACM (2017)
48. Konnov, I.V., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. Inf. Comput. **252**, 95–109 (2017)
49. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 85–102. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_6
50. Krenický, R., Ulbrich, M.: Deductive verification of a Byzantine agreement protocol. Technical report 2010-7. Karlsruhe Institute of Technology, Department of Computer Science (2010)

51. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2004)
52. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
53. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
54. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
55. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
56. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the pastry protocol using TLA$^+$. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 244–258. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21461-5_16
57. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
58. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC 1987, pp. 137–151. ACM (1987)
59. Malkhi, D., Reiter, M.K.: Byzantine quorum systems. In: STOC 1997, pp. 569–578. ACM (1997)
60. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215–226. North-Holland/Elsevier (1989). Reprinted. In: Yang, Z., Marsland, T.A. (eds.) Global States and Time in Distributed Systems, pp. 123–133. IEEE (1994)
61. Méry, D., Singh, N.K.: Automatic code generation from event-B models. In: Symposium on Information and Communication Technology, SoICT 2011, pp. 179–188. ACM (2011)
62. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
63. Newcombe, C.: Why Amazon chose TLA$^+$. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 25–39. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_3
64. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
65. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri Nets, event structures and domains, Part I. Theor. Comput. Sci. **13**, 85–108 (1981)
66. nocrypto. https://github.com/mirleft/ocaml-nocrypto
67. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, 19–20 June 2014, pp. 305–319. USENIX Association (2014)
68. Owre, S., Rushby, J.M., Shankar, N., von Henke, F.W.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. IEEE Trans. Softw. Eng. **21**(2), 107–125 (1995)
69. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI 2016, pp. 614–630. ACM (2016)
70. Panangaden, P., Taylor, K.: Concurrent common knowledge: defining agreement for asynchronous systems. Distrib. Comput. **6**(2), 73–93 (1992)

71. Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. In: SCP (2017)
72. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
73. Schiper, N., Rahli, V., van Renesse, R., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: DSN 2014, pp. 395–406. IEEE (2014)
74. Schmid, U., Weiss, B., Rushby, J.M.: Formally verified Byzantine agreement in presence of link faults. In: ICDCS, pp. 608–616 (2002)
75. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. In: POPL 2018 (2018)
76. Sousa, P.: Proactive resilience. Ph.D. thesis. Faculty of Sciences, University of Lisbon, Lisbon, May 2007
77. Summers, A.J., Müller, P.: Actor services. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 699–726. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_27
78. Tauber, J.A.: Verifiable compilation of I/O automata without global synchronization. Ph.D. thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA (2004)
79. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. **4**(2), 180–209 (1979)
80. Tsuchiya, T., Schiper, A.: Model checking of consensus algorithm. In: SRDS 2007, pp. 137–148. IEEE Computer Society (2007)
81. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 466–480. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87779-0_32
82. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Veríssimo, P.: Efficient Byzantine fault-tolerance. IEEE Trans. Comput. **62**(1), 16–30 (2013)
83. Vukolic, M.: The origin of quorum systems. Bull. EATCS **101**, 125–147 (2010)
84. Wilcox, J.R., Sergey, I., Tatlock, Z.: Programming language abstractions for modularly verified distributed systems. In: SNAPL 2017. LIPIcs, vol. 71, pp. 19:1–19:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
85. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI 2015, pp. 357–368. ACM (2015)
86. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: CPP 2016, pp. 154–165. ACM (2016)